Programming Assignment #2: Single Layer Maze Router

MOHIT SHARMA

110031631

Routing

Routing is the process of generating wiring to interconnect pins of the same signal, while obeying the manufacturing design rules.

Routing is a very complex combinatorial problem whih is usually solved by use of a two-stage approach of global routing followed by detailed routing.

Global routing first partitions the routing region into tiles and decides tile-to-tile paths for all nets while attempting to optimize some given objective function (e.g., total wirelength and circuit timing). Then, guided by the paths obtained in global routing, detailed routing assigns actual tracks and vias for nets.

Problem Definition:

The problem definition for the general routing problem is as follows:

Inputs:

1. A given placement with fixed locations of chip blocks, pins and pads.

2. A netlist

3. A timing budget for each critical net.

4. A set of design rules for manufacturing process, such as resistance, capacitance, and the wire/via width and spacing of each layer.

Objectives:

* Area (channel width) – minimize congestion

* Wire delays – minimize wire length

* Number of layers (less layers => Less expensive)

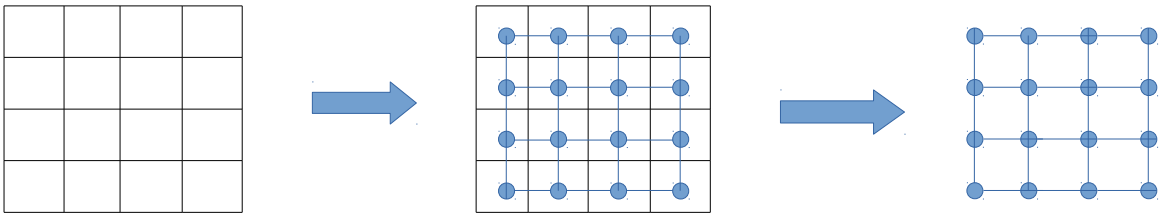
* Additional cost components: Number of bends, vias

Output:

Wire connection for each net presented by actual geometric layout objects that meet the design rules and optimize the given objective.

Routing Model:

Most routing algorithms are based on a graph-search technique guided by the congestion and timing information associated with routing regions and topologies. Applying graph-search technique for routing requires modelling the routing resoure as a graph where the graph topology can represent the chip structure. For modelling, a chip (routing region) is first partitioined into an array of rectangular tiles (or called **global-routing** tiles), each of which may accomodate tens of routing tracks in each dimension. A node in the routing graph represents a tile in the chip, whereas an edge denotes the boundary between two adjacent tiles. Each edge is assigned a capacity according to the physical routing area or the number of tracks in a tile. This graph is called a **global-routing graph.**



a) Paritioned Layout b) Resource Modelling c) Global routing graph

Figure 1: Global routing graph

A global router finds tile-to-tile paths for all nets on the global-routing graph to guide the detailed router. The goal of global routing is to route as many nets as possible while meeting the capacity constraint of each edge and any other constraint, if specified. For example, for timing-driven routing, additional costs can be added to the routing topologies with longer critical path delays. For detailed routing, the router decides the actual physical interconnections of nets by allocating wires on each metal layer and vias for switching between metal layers.

Generally, there are two different layer models, the reserved and unreserved layer models. In the reserved layer model, each layer is allowed only one specific routing direction (i.e., preferred direction). A layer model is unreserved if it allows the placement of wires with any directions (i.e., non-preferred direction). Most of the existing routers and design methodologies apply the reserved layer model, because it has lower complexity than the unreserved layer model and is much easier for implementation.

There are two kinds of detailed-routing models: the grid-based and gridless models. For grid-based routing, a routing grid is superimposed on the routing region, and then the detailed router finds routing paths in the grid. The space between adjacent grid lines is called wire pitch, which is defined in the technology file and is larger than or equal to the sum of the minimum width and

spacing of wires. The router has to control the searching space such that the path in the horizontal/vertical layers can only run horizontally/vertically for the reserved layer model, and switching from layer to layer is allowed only at the intersection of vertical and horizontal grid lines. In this way, the wires with the minimum width following the path in the grid would automatically satisfy the design rules. Therefore, grid-based detailed routing is much more efficient and easier for implementation.

The gridless detailed routing model (also called shaped-based) refers to any model that does not follow the grid-based model. A gridless detailed router does not follow the routing grid and thus can use different wire widths and spacing, Various gridless models have been proposed, such as the connection graph, the implicit connection graph, the implicit triple-line graph, and corner stitching. The main advantage of gridless routing lies in its greater flexibility; it can handle variable widths and spacing for wires and is, thus, more suitable for interconnect tuning optimization, such as wire sizing and perturbation. However, gridless detailed routing is generally much slower than the grid-based one because of its higher complexity.

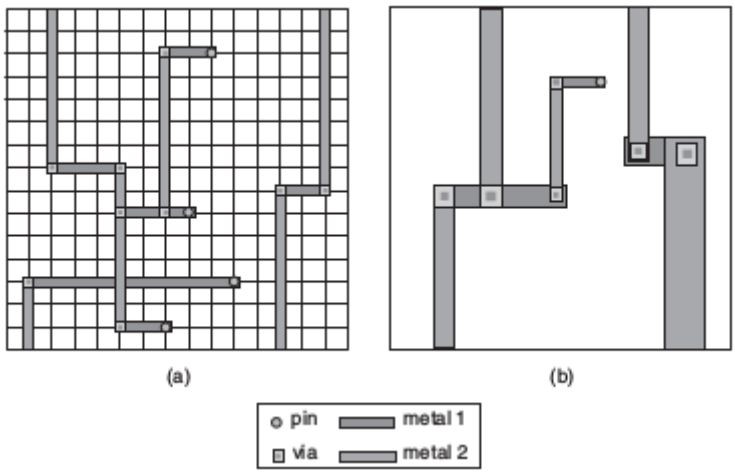


Figure 2: Two kinds of detailed-routing models:
(a) Grid-based detailed routing.
(b) Gridless detailed routing

Routing constraints:

The routing constraints can be classified into two major categories: (1) design-rule constraints and (2) performance constraints.

The design-rule constraint is related with the manufacturing details during fabrication. To improve the manufacturing yield, connections of nets have to follow the rules provided by foundries.

The objective of the performance constraint is to make the connections meet the performance specifications provided by chip designers.

Global vs. Detailed Routing

Global routing

Input to global routing is a detailed placement, with exact terminal locations.

Global routing dertermines “channel” (routing region) for each net while minimizing area and timing.

Detailed routing

Input to detailed routing are channels and approximate routing fro the global routing phase. Detailed routing determines the exact route and layers for each net while minimizing area, meeting timing constraints, number of via's and power to give a valid routing.

General-Purpose routing

For global as well as detailed routing, one can perform a graph-search technique for routing the nets. Popular graph-search techniques include **maze**, **line-search** and **A* search** routing algorithms.

These algorithms are general-purpose routing algorithms, because they can be applied to both global and detailed routing problems on the general routing structure.

Maze routing

Maze routing algorithm is the most widely used algorithm for finding a path between two points. Its also called Lee's algorithm which is based on the breadth-first search (BFS) technique.

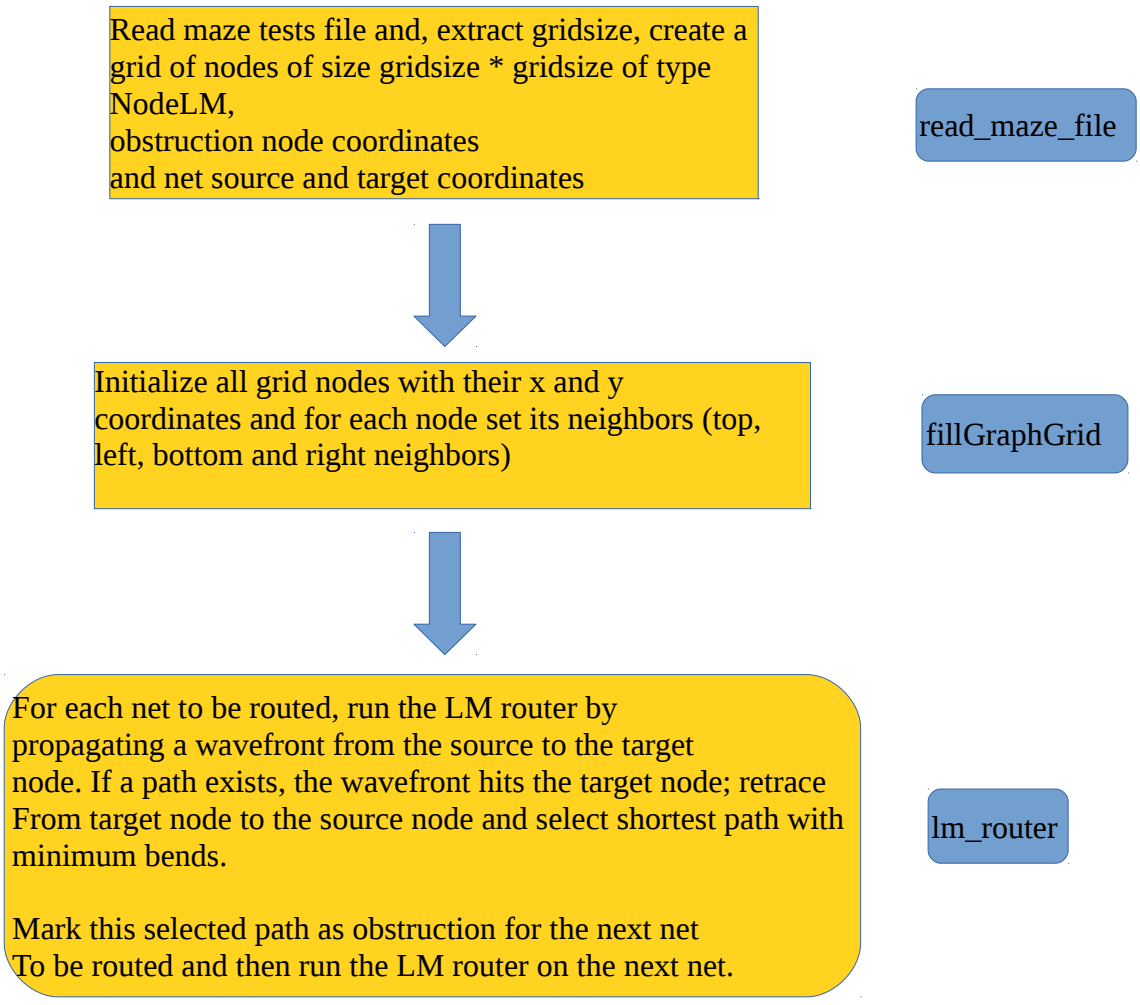
Maze routing adopts a two-phase approach of filling followed by retracing. The filling phase works in the “wave propagation” manner. Starting from the source node S, the adjacent grid cells are progressively labeled one by one according to the distance of the “wavefront” from S until the target node T is reached. Once the target node T is reached, a shortest path is then retraced from T to S with decreasing labels during the retracing phase. Note that any such a path with decreasing labels gives a shortest path. However, we often prefer the one with the least detours for other practical concerns such as the number of bends (vias).

A nice property of Lee's algorithm is that it guarantees to find a path between two points if such a path does exist, and the path is the shortest one, even with obstacles. In practice, however, Lee's algorithm is slow and memory consuming. It has the time and space complexity of O(mn), where

m and n are the respective numbers of horizontal and vertical grid cells. Consequently, it is difficult to apply for large-scale dense designs directly. Because of the pervasive use of Lee's maze-routing algorithm and its high time and space complexity, many methods have been proposed to reduce its running time and memory requirements. These popular optimization methods can be classified into three major categories: (1) coding scheme, (2) search algorithm, and (3) search space.

Maze routing implementation

Flow of program:



The program made as part of the implementation of maze router primarily consists of three function calls:

The read_maze file functions reads the grid size, the obstruction nodes and the source and targeti nodes for the net to be routed. It also creates an array of nodes of type NodeLM which stores the information about nodes such as whether its a target or source node, whether it is the obstruction node, whether it is on a nets selected path etc.

This array of nodes is then returned by the function to the main function. The next function call is to fillGraphGrid function which sets the x and y coordinates of all the nodes, which is used while drawing the nodes when the routing actually occurs, and for each node sets the neighboring nodes as a vector in the _neigbours private member of the node.

The main function then calls the lm_router function which is the actual function that executes the LM router for each net to be routed. While it routes each net, it draws the graphics for the same. i.e. it propagates the wave front from the source node to the target node by labelling each neighboring node in the plist. After the wavefront hits the target node, it sets the path_exists = 1. If path_exists between source and target node, retracing is done from target node to source node via the decreasing labels. The is shown by green connecting lines directed from target to source node. Of the possible retraced paths, the path with minimum bends is selected and a yellow line is drawn to show the same. This path is then marked as an obstruction for the next net to be routed which is routed in a similar manner. If a path does not exist between the source and the target node, the nodes are blinked 5 times, and the order of routing of nets is updated such that this failed net is routed first and the remaining nets are then routed according to the existing order.

The number of times a set of nets is routed is equal to the number of nets being routed.

Main functions:

In file read_data_helper.h:

Reading maze_tests.txt files:

NodeLM* read_maze_file(std::string maze_file, int * gridsize, std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>> *nets, int grid_node_size, int debug)

Inputs:

maze_file: String representing relative or absolute maze file path
*gridsize: Pointer to grid size of integer type
*nets: Pointer to vector of pair of source and target coordinates each of int type
grid_node_size: Int type Node size to draw in graphics
debug: Variable to decide if debug information is to be printed or not.

Outputs:

Returns a pointer to the list of nodes of type NodeLM representing the graph grid.
Appends to vector of pairs the source and target coordinates of nets.
Sets the grid size and the obstruction nodes

Function:

Opens and reads the maze file line by line. When line contains the keyword “grid size” in comments, reads the integer present in the line as the gridsize. When the line contains the keyword “obstruction”, reads two integers representing the x and y coordinates of the node with obstruction. The function sets the private member _is_obstruction to 1 for the node containing the obstruction. When the line contains the keyword “net”, reads four integers representing the x and y coordinates of the source and target nodes. Appends to the vector of pairs i.e. nets the source and target coordinates as a pair. Closes the maze file and returns the nodeList array.

In file LM_graphHelper.h:

Retrieving a triangle for an arrowhead as a CircleShape object:

sf::CircleShape Triangle(float x1, float y1, float x2, float y2, float radius)

Input:

x1: x coordinate of the tail node of the arrowhead.
y1: y coordinate of the tail node of the arrowhead.
x2: x coordinate of the head node of the arrowhead.
y2: y coordinate of the head node of the arrowhead.
radius: Radius of the circumcircle of the triangle.

Output:

Returns a circleshape object with only 3 points so the figure shows up as a regular triangle.

Function:

Create a Circleshape object with three points so create a triangle in a direction from tail node of the arrowhead to the head node of the arrowhead. The size of the triangle is determined by the radius of the circumcircle.

Retrieving a line as a RectangleShape object:

sf::RectangleShape Line(float x1, float y1, float x2, float y2, float thickness = 1)

Input:

x1: x coordinate of the first point on the line
y1: y coordinate of the first point on the line
x2: x coordinate of the second point on the line
y2: y coordinate of the second point on the line
thickness: Thickness of the line.

Output:

Returns a RectangleShape object of given length eqaul to distance between two points and width equal to thickness of the line.

Function:

Creates a RectangleShape object with length equal to distance between the two points and width equal to thickness of the line. The angle of line is determined by $\tan^{-1}$ (180* (y2 – y1)) / (pi * (x2 – x1)). The position of rectangle is set at x1, y1

Draw text

void drawtext(float x, float y, int charsize, sf::Color txt_color, sf::Font *font, std::string txt_string, sf::RenderWindow *window)

#Input:

x: x coordinate of position where to place text.
y: y coordinate of position where to place text.
charsize: Font size of the text.
txt_color: Color of the text.
*font: Pointer to font object in which to draw text
txt_string: String to be shown
*window: Pointer to window object in which to draw.

#Output:

Draws the text in the window.

#Function:

The txt_string is drawn at the x,y coordinate with the given font size, font color, font name on the window.

Draw source and target nodes for a net on the grid

void drawGraph(int n_nodes, NodeLM * nodeList, int grid_node_size, int gridsize, NodeLM s, NodeLM t)

#Input:

n_nodes: Number of nodes in the grid.
*nodeList: Pointer to List of NodeLM objects used to draw the graph grid.
grid_node_size: Size of the square node of the grid.
gridsize: Number of nodes in one side length of the square grid.
s : NodeLM type node representing the source node of the net.
t : NodeLM type node representing the target node of the net.

#Output & Function:

Draws the source node, target node on a square grid of square nodes.

Initialize node x-y coordinates and its neighbors

void fillGraphGrid (NodeLM* nodeList, int gridsize, int grid_node_size)

#Input:

*nodeList: Pointer to List of NodeLM objects used to draw the graph grid.
gridsize: Number of nodes in one side length of the square grid.
grid_node_size: Size of the square node of the grid.

#Output & Function:

Initializes the x-y coordinates of the grid nodes. These x-y coordinates are used to decide the position where to draw the grid nodes. Also, the function initializes the top, bottom, left and right neighbors of the current node (if they exist) in the _neighbors private member of the NodeLM type object.

Initialize RectangleShape grid node parameters

void initialize_nodes_to_draw(sf::RectangleShape *nodesqrs, NodeLM *nodeList, sf::Text *text, sf::Font *font, int n_nodes, int grid_node_size)

#Input:

*nodesqrs: Pointer to array of sf::RectangleShape objects representing the grid nodes.
*nodeList: Pointer to List of NodeLM objects used to draw the graph grid.
*text: Pointer to list of sf::text objects representing the labels to draw while propagating the wavefront
*font: Pointer to font object in which to draw text
*n_nodes: Number of nodes in the grid.
grid_node_size: Size of the square node of the grid.

#Output & Function:

Initializes the sf::RectangleShape node objects, to be drawn,representing the nodes of the square grid by setting its position, outline color, size, outline thickness and fill color. The fill color for a normal node is white while for an obstruction node is black. It also sets the text position, charactersize, color and font for the label.

Draw initial window with boundary nodes, starting grid and the legend

void draw_boundary_nodes_and_starting_grid (sf::RenderWindow *window, sf::RectangleShape *nodesqrs, sf::Font *font, int radius, int gridsize, int grid_node_size, int gridwidth, int gridheight, int n_nodes)

#Input:

*window: Pointer to window object in which to draw.
*nodesqrs: Pointer to array of sf::RectangleShape objects representing the grid nodes.,

*font: Pointer to font object in which to draw text
radius: Radius of the source and target nodes, to be drawn, of nets being routed
gridsize: Number of nodes in one side length of the square grid.
grid_node_size: Size of the square node of the grid.
gridwidth = gridheight = gridsize * grid_node_size
n_nodes: Number of nodes in the grid.

#Output & Function:

Draws the boundary nodes showing the x and y coordinates of the grid. For this purpose, it uses the sf::RectangleShape boundary_object and the drawtext function. Boundary nodes are shown as yellow background square nodes with black text. Also, it draws the square nodes of the grid on which routing will be done. Normally, the grid nodes are white in color unless they are an obstruction in which case they are shown in black color. Then, a line is drawn to the right of the grid dividing the grid from the legend. Legend is drawn such that a green circle is used to represent the current source and target nodes, a blue circle is used to represent other source nodes, a red circle is used to represent other target nodes and a black square is used to represent an obstruction.

Draw nets already routed and all other sources and sinks

void draw_previous_nets_sources_and_sinks (sf::RenderWindow *window, NodeLM *nodeList, int radius, int n_nodes)

#Input:

*window: Pointer to window object in which to draw.
*nodeList: Pointer to List of NodeLM objects used to draw the graph grid.
radius: Radius of the source and target nodes, to be drawn, of nets being routed
n_nodes: Number of nodes in the grid.

#Outpus & Function:

The function is used to draw all the target and source nodes using a circle. A blue circle represents a source node and a red circle represents a target node. The position of the source and target nodes is set at the center of nodes having _has_source_node = 1 or _has_target_node = 1 and is calculated using x and y coordinates of these nodes. Radius of these circles is given by “radius”. A yellow line and a arrow is drawn at the center of the line connecting two nodes as part of an already routed net. Such lines are identified using the “is_to_be_shown_connected_to” NodeLM function.

Draw source and target nodes for current net being routed.

void draw_current_source_and_sink (NodeLM *s, NodeLM *t, int radius, sf::RenderWindow *window, sf::Color source_color, sf::Color target_color)

#Input:

*s: Pointer to source node of current net being routed

*t: Pointer to target node of current net being routed

radius: Radius of the source and target nodes

*window: Pointer to window object in which to draw.

source_color: Color of the source node.

target_color: Color of the target node.

#Output & Function:

Draws the source and target nodes for the current net being routed of the desired color and radius.

Blink the current source and target nodes if their routing was unsuccessful

void blink_current_source_and_sink (NodeLM* s, NodeLM* t, int radius, sf::RenderWindow *window)

#Input:

*s: Pointer to source node of current net being routed

*t: Pointer to target node of current net being routed

radius: Radius of the source and target nodes.

*window: Pointer to window object in which to draw.

#Output and Function

Blinks the source and target nodes for the current net being routed if its routing was unsuccessful.

Draw retraced path

void draw_traced_path(NodeLM *s, NodeLM *t, int ctr, sf::RenderWindow *window)

#Input:

*s: Pointer to source node of current net being routed

*t: Pointer to target node of current net being routed

ctr: Net number being routed

*window: Pointer to window object in which to draw.

#Output & Function:

Draws the selected path with minimum bends, retraced if a path exists.

In file LM_Router.h:

Get direction of second node as compared with the first node

std::string get_dir(int node1_x, int node1_y, int node2_x, int node2_y)

#Input:

node1_x: x coordinate of the first node.
node1_y: y coordinate of the first node.
node2_x: x coordinate of the second node.
node2_y: y coordinate of the second node.

#Output:

Returns “down”, “up”, “left” or “right” depending on where the second node lies w.r.t the first node.

#Function:

Returns the direction in which the second node lies w.r.t the first node.

Retrace path from the target node to the source node if it exists in the order of decreasing labels

void retrace(NodeLM* s, NodeLM* t, int gridsize, int grid_node_size, int ctr, sf::RenderWindow *window)

#Input:

*s: Pointer to source node of current net being routed
*t: Pointer to target node of current net being routed
gridsize: Number of nodes in one side length of the square grid.
grid_node_size: Size of the square node of the grid.
ctr: Net number being routed
*window: Pointer to window object in which to draw.

#Output:

Draws a green line from node with label = label to a node with label = label – 1. This is done starting from the target node via the intermediate nodes with successively decreasing labels untill the source node is reached.

#Function:

The function saves path information node by node starting from the target node. Foreach neighbor of the current node being processed, it adds information to the neighboring node about the current node in a unique identifier referred to as prev_node_data. The data consists of a tuple of following elements:
“Net num being routed”, “Sum of x coordinates of the nodes lying on this path”,

"Sum of y coordinates of the nodes lying on this path", "pointer to previous node", "direction of neighboring node w.r.t. current node" , "number of bends" , "Boolean value specifying whether node lies on multiple net paths", "Boolean value specifying that node lies on already routed path"). It then draws a green line from node with label = label to a node with label = label – 1.

Main pass of Lee-Moore maze router:

void lm_router(std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>> *nets, NodeLM *nodeList, int n_nodes, int gridsize, int grid_node_size)

#Inputs:

*nets: Pointer to vector of pair of source and target coordinates each of int type
*nodeList: Pointer to List of NodeLM objects used to draw the graph grid.
n_nodes: Number of nodes in the grid.
gridsize: Number of nodes in one side length of the square grid.
grid_node_size: Size of the square node of the grid.

#Outputs & Function:

Label's nodes starting from source node untill it hits the target node. If the wavefront hits the target node, the algorithm declares that a path exists between source and target node else path does not exist between source and target nodes. If path exists, retrace the path from target node to source node using the retrace function and draw the traced path using draw_traced_path function.

If path does not exist and num_tries is greater than 0, blink the current source and

target nodes, decrement number of tries, reorder nets to be routed such that the current net is routed first, initialize all variables again as well as the nodes data to begin routing from beginning then draw_boundary_nodes_and_starting_grid and draw_previous_nets_sources_and_sinks. This will initialize the draw window. Restart routing from the first net.

In file src/Lee_Moore_routing.cpp:

Main function for the program

int main(int argc, char * argv[])

#Inputs:

argc: Number of arguments passed to the program

argv: Array of arguments passed to the program

#Outputs and Function:

Executes the Lee Moore router program

Data Structure Used:

The fundamental unit for the LM maze router algorithm in this implementation is the Nod

eLM type object.

NodeLM object

```
class NodeLM: public Node {
    public:
        NodeLM(float x, float y, std::string name = ""): Node(x, y, name) {}
        NodeLM() {}
        void setLabel(int label);
        int getLabel();
        bool is_obstruction();
        void setnodeasobstruction();
        std::vector<NodeLM*> neighbors();
        void addNeighbor(NodeLM* node);
        void update_line_exists_with_neighbor_node(NodeLM*,int);
        void update_has_source_node(int);
        int is_source_node();
        int is_target_node();
        void update_has_target_node(int);
        int is_to_be_shown_connected_to(NodeLM*);
        std::string get_direction(int,NodeLM*,std::string,std::string);
        int get_num_bends(int,NodeLM*,std::string,std::string);
        std::vector<std::pair<std::pair<std::string,std::string>,NodeLM*>>
get_path(int);
        void add_prev_node_data(int, std::string, std::string, NodeLM*,
std::string, int, int, int);
        void delete_prev_node_data();
        int is_already_on_a_nets_path(int);
        int is_already_on_a_selected_path();
        void add_on_selected_path();
        void delete_from_selected_path();
        std::pair<std::string,NodeLM*> get_path_with_min_bends(int,
std::string);
        std::pair<int,int> get_path_on_single_net_and_with_min_bends(int);
        std::pair<int,int> get_coordinates(int);
        int get_node_num(int, int);
    private:
        int _label = 0;
        bool _is_obstruction = false;
        std::vector<std::pair<NodeLM*,int>> _neighbors;
        int _is_on_selected_path;
        int _has_source_node = 0;
        int _has_target_node = 0;
std::vector<std::tuple<int,std::pair<std::string,std::string>,NodeLM*,std::strin
g,int,int>> _path_data;
};
```

NodeLM class object is polymorphised on class Node.
=> It has the following public member functions:

* Two NodeLM constructors which are overloaded. One accepts arguments as x and y coordinates and the object name. The other does not accept any arguments.
* **void** setLabel(**int** label) : Function to set the label for a node.

```
* int getLabel() : Function to get the label for a node.

* bool is_obstruction() : Function to check if a node is obstruction.

* void setnodeasobstruction() : Function to set a node as an obstruction.

* std::vector<NodeLM*> neighbors() : Function get the neighbors for a node.

* void addNeighbor(NodeLM* node) : Function to add a neighbor for a node.

* void update_line_exists_with_neighbor_node(NodeLM*,int) : Function to set or
unset line exists with a neighboring node.

* void update_has_source_node(int) : Function to set or unset a node as source
node.

* int is_source_node() : Function to check if a node is a source node.

* int is_target_node() : Function to check if a node is a target node.

* void update_has_target_node(int) : Function to set or unset a node as target
node.

* int is_to_be_shown_connected_to(NodeLM*) : Function to check if line exists
with a neighboring node.

* std::string get_direction(int,NodeLM*,std::string,std::string) : Function to
get direction of node with respect to a previous node on a path for a net.

* int get_num_bends(int,NodeLM*,std::string,std::string) : Function to get
number of bends on a path for a net.

* std::vector<std::pair<std::pair<std::string,std::string>,NodeLM*>>
get_path(int);

* void add_prev_node_data(int, std::string, std::string, NodeLM*, std::string,
int, int, int) : Function to add previous node data on a path for a net.

* void delete_prev_node_data() : Function to clear the _path_data

* int is_already_on_a_nets_path(int): Function to return check if node already
lies on a nets path.

* int is_already_on_a_selected_path(): Function to return _is_on_selected_path
private member for a node.

* void add_on_selected_path() : Function to set the _is_on_selected_path private
member.

* void delete_from_selected_path() : Function to unset the _is_on_selected_path
private member.
```

```
* std::pair<std::string,NodeLM*> get_path_with_min_bends(int, std::string) : Of all the paths on which the node lies for net, get the path with minimum bends.

* std::pair<int,int> get_coordinates(int) : Get x-y coordinates for a node.
```

* **int** get_node_num(**int**, **int**) : Get node number.

=> It has the following private members:

```
        int _label = 0 : Stores the label for a node set during wavefront propagation.
        bool _is_obstruction = false : Stores whether a node is obstruction node or not.
        std::vector<std::pair<NodeLM*,int>> _neighbors : Stores the neighbors for a node.
        int _is_on_selected_path : Boolean to check if a node is already on a selected path for a net.
        int _has_source_node = 0 : Boolean to store if a node is a source node.
        int _has_target_node = 0 : Boolean to store if a node is a target node.
std::vector<std::tuple<int,std::pair<std::string,std::string>,NodeLM*,std::string,int,int>> _path_data : Tuple to store path data on which the node lies for a net.
```

Node Object:

```
class Node {
   public:
           Node();
           Node(float x, float y, std::string name = "");
           Node(Node* node);
           float x();
           float y();
           std::string name();
           void setX(float x);
           void setY(float y);
           void setName(std::string name);
   private:
           float _x;
           float _y;
           std::string _name;
};
```

Class Node has following public member functions:

* It has the following three constructors that are overloaded. First constructor does not accept any arguments. Second accepts float type x, y coordinates and string type node object name initialized to empty string. Third accepts another node object to initialize the current node object.
* float x() : Retrieves the _x private member of the node.
* float y() : Retrieves the _y private member of the node.
* std::string name() : Retrieves the _name private member of the object
* void setX(float x) : Sets the _x private member of the node.
* void setY(float y) : Sets the _y private member of the node.

* void setName(std::string name) : Sets the _name private member of the objective

Class Node has the following private member functions:

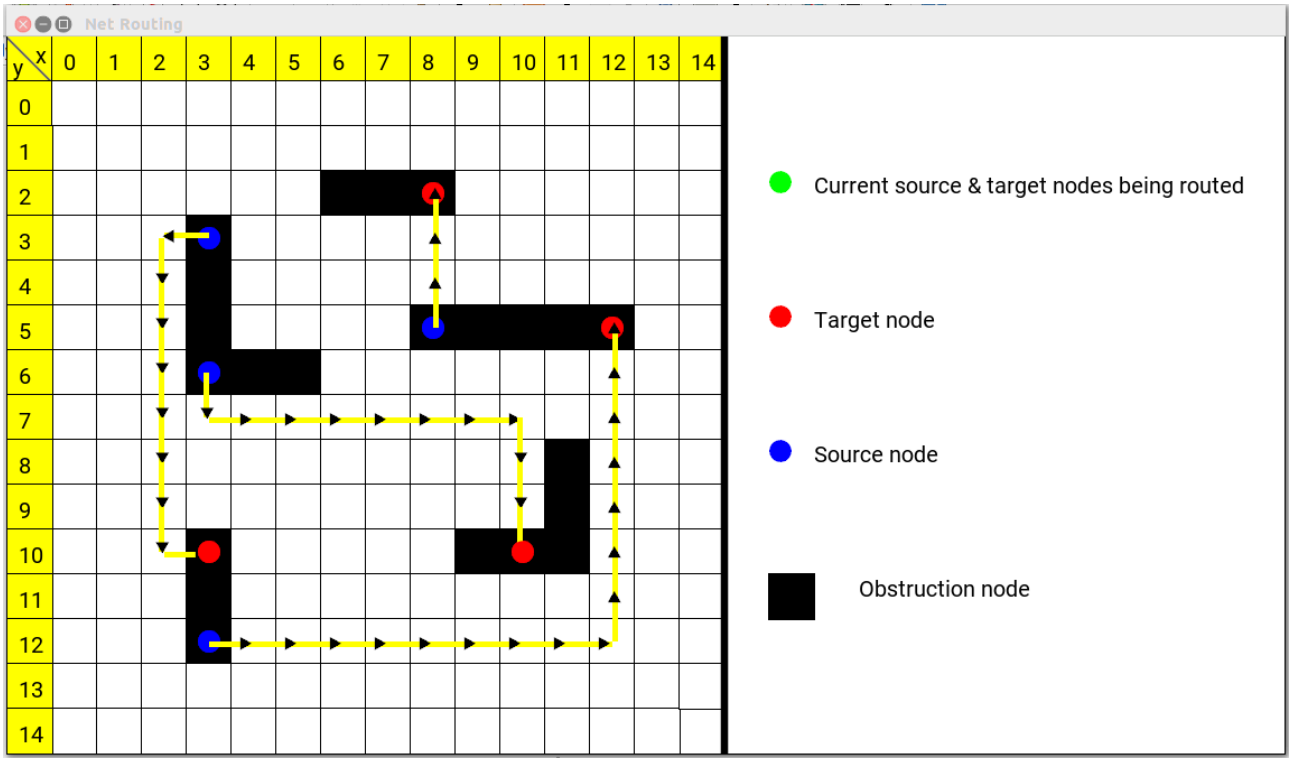
* float _x : Stores the x coordinate of the node. This is used as the x-coordinate of window wherethe node is drawn.
* float_y : Stores the y coordinate of the node. This is used as the y-coordinate of window where the node is drawn.
* std::string _name : Stores the name of the current object.

Results for experimental runs

Following results were obtained with LM maze router algorithm implementation:

Netlist file 1: Gridsize = 15, #Nets to route = 4

Graphical Output:



Shell output:

****
*Try number : 1

**Routing net number: 1

***Source coordinates: 3:12
***Target coordinates: 12:5

Path exists between source and target nodes
_____

**Routing net number: 2

***Source coordinates: 3:6
***Target coordinates: 10:10

Path exists between source and target nodes
_____

**Routing net number: 3

***Source coordinates: 8:5
***Target coordinates: 8:2

Path exists between source and target nodes
_____

**Routing net number: 4

***Source coordinates: 3:3
***Target coordinates: 3:10

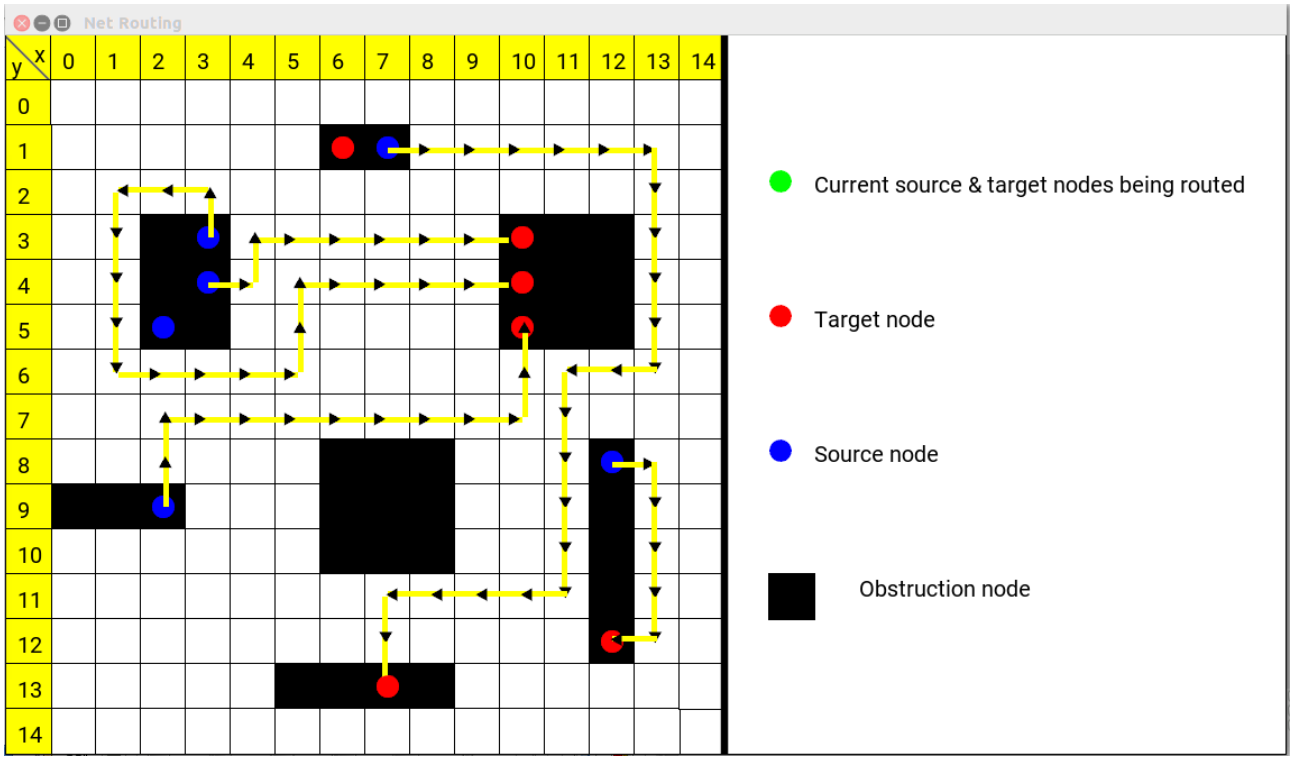
Path exists between source and target nodes
_____

****

****
Number of nets successfully routed = 4/4 in try number 1
****

Netlist file 2: Gridsize = 15, #Nets to route = 6

Graphical Output:



Shell output:

****

*Try number : 1

**Routing net number: 1

***Source coordinates: 3:3

***Target coordinates: 10:4

Path exists between source and target nodes

**Routing net number: 2

***Source coordinates: 7:1

***Target coordinates: 7:13

Path exists between source and target nodes

**Routing net number: 3

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes

**Routing net number: 4

***Source coordinates: 3:4
***Target coordinates: 10:3

Path does not exist between source and target nodes

****
*Try number : 2

**Routing net number: 1

***Source coordinates: 3:4
***Target coordinates: 10:3

Path exists between source and target nodes

**Routing net number: 2

***Source coordinates: 3:3
***Target coordinates: 10:4

Path exists between source and target nodes

**Routing net number: 3

***Source coordinates: 7:1
***Target coordinates: 7:13

Path exists between source and target nodes

**Routing net number: 4

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes

**Routing net number: 5

***Source coordinates: 2:5
***Target coordinates: 6:1

Path does not exist between source and target nodes

****
*Try number : 3

**Routing net number: 1

***Source coordinates: 2:5
***Target coordinates: 6:1

Path exists between source and target nodes

**Routing net number: 2

***Source coordinates: 3:4
***Target coordinates: 10:3

Path exists between source and target nodes

**Routing net number: 3

***Source coordinates: 3:3
***Target coordinates: 10:4

Path exists between source and target nodes

**Routing net number: 4

***Source coordinates: 7:1
***Target coordinates: 7:13

Path exists between source and target nodes

**Routing net number: 5

***Source coordinates: 2:9
***Target coordinates: 10:5

Path does not exist between source and target nodes

****
*Try number : 4

**Routing net number: 1

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes

**Routing net number: 2

***Source coordinates: 2:5
***Target coordinates: 6:1

Path exists between source and target nodes
_____

**Routing net number: 3

***Source coordinates: 3:4
***Target coordinates: 10:3

Path exists between source and target nodes
_____

**Routing net number: 4

***Source coordinates: 3:3
***Target coordinates: 10:4

Path exists between source and target nodes
_____

**Routing net number: 5

***Source coordinates: 7:1
***Target coordinates: 7:13

Path does not exist between source and target nodes

****
*Try number : 5

**Routing net number: 1

***Source coordinates: 7:1
***Target coordinates: 7:13

Path exists between source and target nodes

_____

**Routing net number: 2

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes

_____

**Routing net number: 3

***Source coordinates: 2:5
***Target coordinates: 6:1

Path exists between source and target nodes

_____

**Routing net number: 4

***Source coordinates: 3:4
***Target coordinates: 10:3

Path exists between source and target nodes

_____

**Routing net number: 5

***Source coordinates: 3:3
***Target coordinates: 10:4

Path does not exist between source and target nodes

****
*Try number : 6

**Routing net number: 1

***Source coordinates: 3:3
***Target coordinates: 10:4

Path exists between source and target nodes
_____

**Routing net number: 2

***Source coordinates: 7:1
***Target coordinates: 7:13

Path exists between source and target nodes
_____

**Routing net number: 3

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes
_____

**Routing net number: 4

***Source coordinates: 2:5
***Target coordinates: 6:1

Path exists between source and target nodes
_____

**Routing net number: 5

***Source coordinates: 3:4
***Target coordinates: 10:3

Path does not exist between source and target nodes

****
*Try number : 7

**Routing net number: 1

***Source coordinates: 3:4
***Target coordinates: 10:3

Path exists between source and target nodes

**Routing net number: 2

***Source coordinates: 3:3
***Target coordinates: 10:4

Path exists between source and target nodes

**Routing net number: 3

***Source coordinates: 7:1
***Target coordinates: 7:13

Path exists between source and target nodes

**Routing net number: 4

***Source coordinates: 2:9
***Target coordinates: 10:5

Path exists between source and target nodes

**Routing net number: 5

***Source coordinates: 2:5
***Target coordinates: 6:1

Path does not exist between source and target nodes

****

**Routing net number: 6

***Source coordinates: 12:8

***Target coordinates: 12:12

Path exists between source and target nodes

_____

****

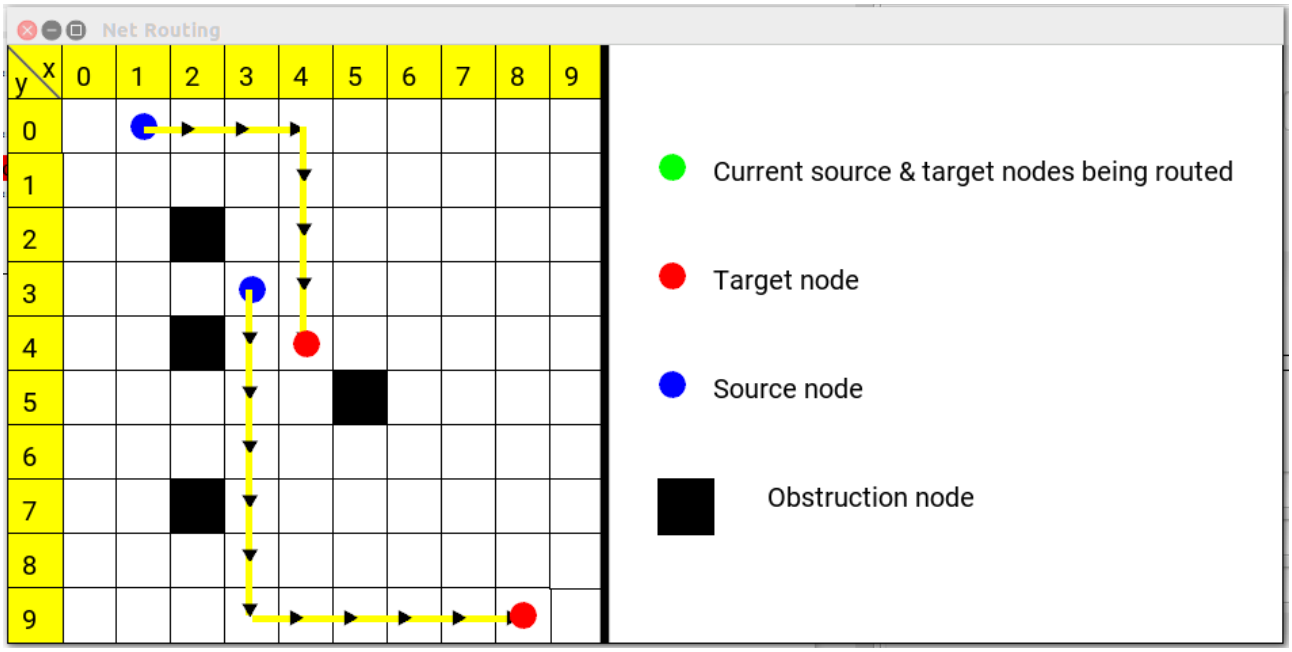
****

Number of nets successfully routed = 5/6 in try number 7

****

Netlist file 3: Gridsize = 10, #Nets to route = 2

Graphical Output:



Shell output:

```
**********************************************************************
*Try number : 1

**Routing net number: 1

----------------------------
***Source coordinates: 1:0
***Target coordinates: 4:4
----------------------------
Path exists between source and target nodes
_____________________________________________________

**Routing net number: 2

----------------------------
***Source coordinates: 3:3
***Target coordinates: 8:9
----------------------------
Path exists between source and target nodes
_____________________________________________________

**********************************************************************

**********************************************************************
Number of nets successfully routed = 2/2 in try number 1
**********************************************************************
```

References

* Electronic Design Automation: Synthesis, Verification, and Test by Laung-Terng Wang, Yao-Wen Chang, Kwang-Ting (Tim) Cheng

* https://github.com/abangfarhan/graph-sfml

* Lecture Notes: Physical Design Automation for VLSI and FPGAs: Routing

---